Práctica 2

Búsqueda no Informada

(Búsqueda primero en anchura y Búsqueda primero en profundidad)

RESUMEN

Se realizó la implementación de los algoritmos de búsqueda no informada primero en anchura y primero en profundidad sobre el problema del clásico juego de 15-puzzle con el objetivo de comparar ambos algoritmos en la resolución de este problema, como sabemos en base a la teoría la búsqueda primero en anchura es un algoritmo más completo pero lento y el algoritmo primero en profundidad tiende a ser más rápido, pero no óptimo.

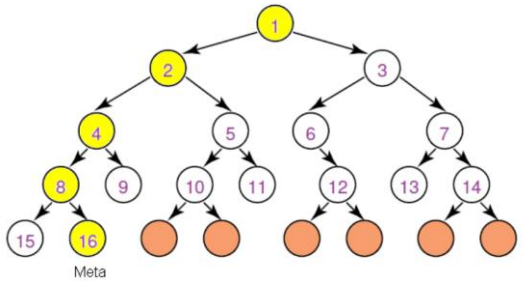
INTRODUCCION

Búsqueda no Informada

Dentro de las técnicas de búsqueda empleadas por los agentes encontramos dos tipos de búsqueda, la informada y la no informada, en la búsqueda no informada no se cuenta con información adicional acerca de los estados más allá de la proporcionada por la definición del problema, por tanto, su proceso se basa en generar diferentes estados, haciendo comparaciones con el estado objetivo hasta llegar a él.

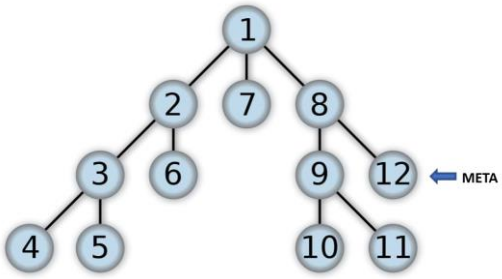
Búsqueda primero en anchura (BFS)

En esta estrategia primero se expande el nodo raíz y a continuación los nodos sucesores y así nivel a nivel, de tal forma que antes de expandir en profundidad se expande por completo cada nivel del árbol de búsqueda.



Búsqueda primero en profundidad (DFS)

A diferencia de la anterior, en esta búsqueda se analiza por completo cada rama del árbol hasta encontrar la solución o llegar a un nodo hoja, es decir, sin nodos hijos, en caso de que esto ocurra se regresa al ascendente inmediato y continúa explorando los nodos hijos sin explorar, en definición podría ser considerado Back-tracking.



DESARROLLO DE LA PRÁCTICA

La práctica se desarrolló utilizando el lenguaje de programación Java y sus capacidades de Programación Orientada a Objetos.

El programa consta de 4 clases las cuales son Pozole (la clase principal), Tablero (clase que instancia y muestra la interfaz gráfica), State (que almacena los estados del programa durante la ejecución) y por último Executor (que ejecuta un hilo que muestra al usuario como sería la resolución paso a paso de manera gráfica).

Pozole.java (Main Class)

La clase principal solamente instancia a la clase tablero y la hace visible.

```java
package pozole;

import java.util.ArrayList;
import java.util.ArrayDeque;

public class Pozole {

    /**
     * @param args the command line arguments
     */
    Run | Debug
    public static void main(String[] args){
        Tablero t =new Tablero();
        t.setVisible(b: true);
    }
}
```

Tablero.java (Clase de Interfaz Gráfica)

La clase tablero es la que gestiona y prácticamente administra el programa de cara al usuario implementa la interfaz gráfica y la cinta de opciones del sistema y muestra al usuario las opciones del sistema.

Tenemos primeramente una función que inicializa las variables y los componentes de la interfaz gráfica para poder utilizar los recursos del sistema.

```java
private void initComponents(){ //Inicializamos los componenentes
    this.setTitle(title: "15-Puzzle"); //Establecemos el titulo de nuestra ventana
    this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE); //Indicamos al programa que no haga nada o ignore el valor de exit predefinido

    Dimension pantalla = Toolkit.getDefaultToolkit().getScreenSize(); //Obtebemos los valores de la pantalla
    int width = pantalla.width; //Obtenemos el ancho
    int height = pantalla.height; //Obtenemos el alto
    this.setBounds((width-516)/2,(height-563)/2,width: 516,height: 563); //Enviamos la posicion de la ventana al sistema para posicionar el valor

    JMenuBar mainMenu = new JMenuBar(); //Declaramos nuestra barra de menu
    JMenu    file = new JMenu(s: "File"); //Declaramos la opcion file del menu
    JMenuItem exit = new JMenuItem(text: "Exit"); //Declaramos la opcion exit del menu

    mainMenu.add(file); //Añadimos file al menu barra principal
    file.add(solveB); //Añadimos nuestras opciones del submenu file
    file.add(solveD);
    file.add(exit);
    this.setJMenuBar(mainMenu); //Añadimos nuestro menu de barra a nuestro programa

    this.setLayout(manager: null);
    this.imagePieces(pathName: "imagenes/gato_redim.jpg"); //Asignamos la ruta de nuestra imagen que mostraremos en el puzzle
    paintPieces(); //Llamada a la función que mostrará en la UI nuestro puzzle con sus respectivas imagenes
    exit.addActionListener(evt -> gestionarExit(evt));  //Asignamos el vento de salida
    solveB.addActionListener(evt -> whichMethod(evt));  //Asignamos el evento de solucion
    solveD.addActionListener(evt -> whichMethod(evt));  //Asignamos el evento de solucion

    // Handle the X-Button
    class MyWindowAdapter extends WindowAdapter{
        @Override
        public void windowClosing(WindowEvent eventObject){
            goodBye();
        }
    }
    addWindowListener(new MyWindowAdapter());
}
```

Para poder gestionar esta práctica estaremos trabajando con estructuras de datos (cola y pila), hilos de ejecución, matrices y elementos gráficas de las mismas librerías de Java.

Como parte del sistema gestionamos la salida del sistema de manera personalizada con un mensaje especifico.

```
83  private void goodBye(){ //Nuestra funcion de salida
84      int respuesta = JOptionPane.showConfirmDialog(rootPane, message: "Are you sure?",title: "Exit",JOptionPane.YES_NO_OPTION);
85      //Preguntamos al usuario si esta seguro de salir del programa
86      if(respuesta==JOptionPane.YES_OPTION) System.exit(status: 0); //Con base en la respuesta damos la salida dels sistema
87  }
88
89  private void gestionarExit(ActionEvent e){
90      goodBye(); //Asignamos nuestro evento de salida
91  }
92
```

Y utilizamos dos funciones para poder mostrar al usuario la representación gráfica de nuestro juego 15-puzzle. La primera función se encarga de dividir y almacenar los valores de cada subimagen en un *HashMap* tomando como clave el valor de nuestro estado objetivo.

```
94  // Parte la imagen en piezas
95  private void imagePieces(String pathName){
96      // Bloque "try" por que hay una tarea de lectura de archivo
97      try{
98          BufferedImage buffer= ImageIO.read(new File(pathName));
99          BufferedImage subImage;
100         int n=0;
101         for(int i=0;i<4;i++)
102             for(int j=0;j<4;j++){
103                 subImage = buffer.getSubimage((j)*125, (i)*125, w: 125, h: 125);  // Extrae un fragmento de la imagen
104                 String k = goal.substring(n,n+1);
105                 puzzle.put(k, subImage); // Almacena las piezas etiquetándolas con base en el estado final
106                 n++;
107             }
108     }
109     catch (Exception ex){
110         ex.printStackTrace();
111     }
112
113 }
```

Por su parte la función para “pintar” el tablero lo hace con base en el estado inicial. Tanto el estado inicial, como el estado objetivo se definen como una cadena de caracteres en las variables de entorno de la clase.

```
115 public void paintPieces(){
116     int n = 0;
117     for(int i = 0; i < 4; i++)
118         for(int j = 0; j < 4; j++){
119             String  k = start.substring(n, n+1);
120             BufferedImage subImage = (BufferedImage) puzzle.get(k);
121             jBoard[i][j] = new JButton();
122             jBoard[i][j].setBounds(j*125+1, i*125+1, width: 125, height: 125); // Calcula la posición del botón i,j
123             this.add(jBoard[i][j]);
124             if(!k.equals(anObject: "0"))jBoard[i][j].setIcon(new ImageIcon(subImage));
125             else empty = subImage;
126             n++;
127         }
128
129 }
```

En el menú del sistema damos la opción a seleccionar por cual método se buscará la solución al problema y con la siguiente función validamos el método.

```
private void whichMethod(ActionEvent e){
    if(e.getSource() == solveD) deep = true;  // En caso de que se trate de búsqueda en profundidad
    solve();
}
```

Ahora bien, la parte más fundamental del sistema es implementar los algoritmos de búsqueda, de manera directa ambos algoritmos se implementan de manera muy similar solo diferenciando en la manera de operar la estructura de datos pues mientras para la búsqueda en anchura utilizamos una cola, para la búsqueda en profundidad nos apoyamos de una pila.

A continuación, presentamos el método solve:

```java
private void solve(){
    boolean success = false; //declaramos un booleano que nos indicará el exito
    int deadEnds = 0;
    int totalNodes = 0;
    State startState = new State(start);
    State goalState   = new State(goal);
    ArrayDeque queue = new ArrayDeque();
    ArrayList<State> first = new ArrayList();
    ArrayList<State> path=null;
    solveB.setEnabled(b: false);
    solveD.setEnabled(b: false);

    first.add(startState);
    queue.add(first);

    // Search loop
```

```java
    int m=0;
    long startTime = System.currentTimeMillis();
    while(!queue.isEmpty() && !success && m < maxDeep ){
        int validStates = 0;
        m++;
        //System.out.println("Ciclo " + m);
        ArrayList<State> l = (ArrayList<State>) queue.getFirst();
        //System.out.println("Analizando Ruta de :" + l.size());
        // muestraEstados(l);
        State last = (State) l.get(l.size()-1);
        //last.show();
        ArrayList<State> next = last.nextStates();
        //System.out.println("Se encontraron " + next.size()+ " estados sucesores posibles");
        totalNodes += next.size();

        queue.removeFirst(); // Se elimina el primer camino de la estrutura

        for(State ns: next){
            if(!repetido(l,ns)){ // Se escribió un método propio para verificar repetidos
                validStates++;
                ArrayList<State> nl = (ArrayList<State>) l.clone();
                if(ns.goalFunction(goalState)){
                    success = true;
                    path = nl;
                }
                nl.add(ns);
                //muestraEstados(nl);
                if(deep) queue.addFirst(nl); // Si es en profundidad agrega al principio la nueva ruta
                else queue.addLast(nl); // Si es en anchura agrega al final
                //System.out.println("Agregé un nuevo camino con "+nl.size()+ " nodos");
            }
            //else System.out.println("Un nodo repetido descartado");
        }
        if(validStates==0) deadEnds++;  // Un callejón sin salida
    }
```

```java
    if(success) // Si hubo éxito
    {
        long elapsed = System.currentTimeMillis()-startTime;
        if(deep) this.setTitle(title: "15-Puzzle (Deep-First Search)");
        else this.setTitle(title: "15-Puzzle (Breadth-First Search)");
        JOptionPane.showMessageDialog(rootPane, "Success!! \nPath with "+path.size()+" nodes"+"\nGenerated nodes: "+totalNodes+"\nDead ends: "+ deadEnds+"\nLoops: "+m
                                      title: "Good News!!!", JOptionPane.INFORMATION_MESSAGE);
        System.out.println(x: "Success!");
        String thePath="";
        int n=0;
        int i=startState.getI();
        int j=startState.getJ();
        for(State st: path)
        {
          st.show();
          if(n>0)
            thePath = thePath+st.getMovement();
          n++;
        }
        Executor exec = new Executor(jBoard,i,j,thePath, empty);
        exec.start();
    }
    else
    {
        JOptionPane.showMessageDialog(rootPane, message: "Path not found", title: "Sorry!!!", JOptionPane.WARNING_MESSAGE);
        System.out.println(x: "Path not found");
    }
}
```

Ahora bien, cabe resaltar que tal y como se comenta en las líneas del código, hubo necesidad de diseñar un algoritmo personalizado para comparar o verificar la equivalencia entre estados. Dicho método se ve de la siguiente manera:

```java
// Compara para evitar nodos repetidos
public boolean repetido(ArrayList<State> l, State s)
{
    boolean exist = false;
    for(State ns: l)
    {
        if(ns.isEqual(s)) // Un método propio para compaarar estados
        {
            exist = true;
            break;
        }
    }
    return exist;
}
```

State.java

La clase State tal cual se encarga de generar objetos de tipo estado que almacenen los estados por los que va atravesando el algoritmo durante su ejecución o visto desde otra manera, cada uno de los nodos del árbol de búsqueda que se va generando.

Sus atributos principales son el tablero, la posición en x, la posición en y, el movimiento a realizar.

Tenemos desde luego el método constructor y los *getter* de los atributos, así como algunos otros métodos que son utilizados durante la ejecución del algoritmo, como lo es el método show que muestra el estado del objeto. El método swap que intercambia o representa un intercambio de estados.

Executor.java

La clase Executor de nuestro programa controla el hilo del sistema con el objetivo de mostrarle a nuestro usuario de manera gráfica como sería el movimiento de las fichas para dar solución al puzzle.

La clase cuenta con el método constructor que maneja los atributos de tablero de botones, posición en i, posición en j, la ruta o camino de solución, un valor *empty* que almacena la subimagen de la posición 0.

Y la clase run que ejecuta el hilo y le da el funcionamiento de mover las fichas de manera gráfica de acuerdo con los movimientos de la ruta de solución encontrada por nuestro algoritmo de búsqueda no informada.

Considerando que la cantidad de nodos que tendrá la ramificación de solución puede tender a ser muy amplia, el tiempo de vida del hilo operará acorde a la cantidad de movimientos que tiene o tendrá que realizar para darle solución al problema de manera visual:

```
public void run(){
    for(int n=0;n<path.length();n++){
            int newI=i;
            int newJ=j;
            char m=path.charAt(n);
            switch(m){
                case 'u' -> newI--;
                case 'd' -> newI++;
                case 'l' -> newJ--;
                case 'r' -> newJ++;
            }
            try{
                Thread.sleep((int) (10000/(path.length()-1)));  // El tiempo de sleep en función del número de movimientos
            }
            catch(Exception ex){ ex.printStackTrace();}
            Icon sw = jBoard[newI][newJ].getIcon();

            jBoard[newI][newJ].setIcon(defaultIcon: null);
            jBoard[i][j].setIcon(sw);
            i=newI;
            j=newJ;

        }
        jBoard[i][j].setIcon(new ImageIcon(empty));
}
```

Demostración de Funcionamiento

Cabe resaltar que utilizaremos dos estados iniciales diferentes para cada método de solución pues puede tender a complicarse o extenderse mucho la búsqueda por profundidad dependiendo de la solución encontrada.

Pantalla principal

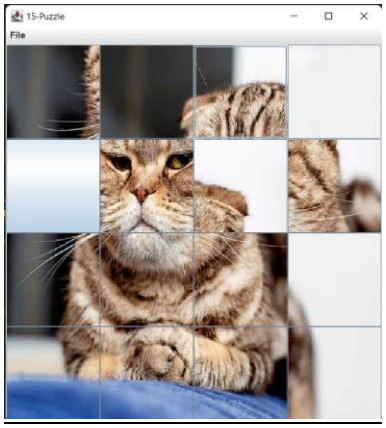


Ilustración 1. Pantalla de inicio búsqueda por anchura

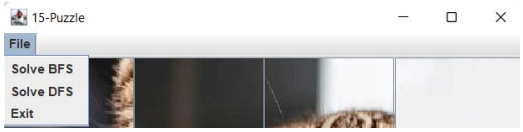


Ilustración 2. Menú de opciones

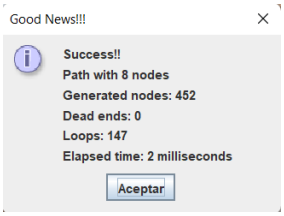


Ilustración 3. Mensaje de éxito

Ilustración 4. Puzzle resuelto

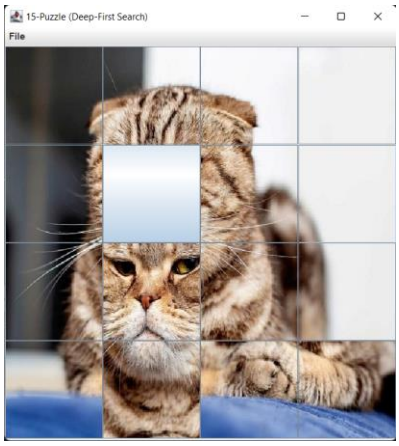


Ilustración 5. Pantalla inicial para búsqueda en profundidad

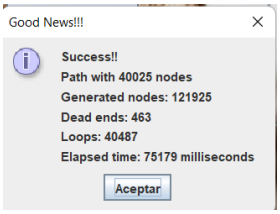


Ilustración 6. Mensaje de éxito

Ilustración 7. Puzzle resuelto

Conclusiones

Los algoritmos de búsqueda no informada al no tener más información más allá de la provista por el planteamiento del problema, es decir, un estado inicial y un estado final, así como ciertas reglas o condiciones del juego, tienden a tener dos conflictos con las soluciones que proponen, y es que pueden convertirse en algoritmos muy pesados de una complejidad elevada o pueden llegar a soluciones no óptimas. Sin embargo, esto no quita que sean una gran herramienta cuando nuestras condiciones son tal cuál no contar con más información que la plantea en un inicio.